

Εργαστήριο Λογικής Σχεδίασης

5^η Διάλεξη

VHDL-Τύποι σημάτων-Process

Βασιλόπουλος Διονύσης

Ε.Δι.Π. Τμήματος Πληροφορικής & Τηλεπικοινωνιών

VHDL – Τύποι σημάτων

Signed-Unsigned αριθμοί

Για να χρησιμοποιήσουμε αριθμούς (προσημασμένους ή μη) πρέπει να δηλώσουμε το πακέτο/βιβλιοθήκη `numeric_std` με τη δήλωση

```
use IEEE.NUMERIC_STD.ALL;
```

Εάν θέλουμε να δηλώσουμε ένα σήμα ως αριθμό τότε η δήλωση

```
signal x: signed(7 downto 0);
```

Αναφέρεται σε **προσημασμένο αριθμό των 8 bit συμπληρώματος ως προς 2**

ενώ η δήλωση

```
signal x: unsigned(7 downto 0);
```

Αναφέρεται σε **μη προσημασμένο αριθμό των 8 bit**

Δηλώνονται ως διανύσματα όπως και ο τύπος `std_logic_vector`

Επιτρέπουν την **εκτέλεση αριθμητικών πράξεων** σε αντίθεση με τα διανύσματα `std_logic_vector`

VHDL – Unsigned

Πράξεις

Έστω όλα τα σήματα unsigned(3 down to 0), 4 bits

`a<="1000"; = 8`

`b<="0001"; = 1`

Οι τιμές δίνονται όπως και σε `std_logic_vector`

`add<=a+b;`

`sub<=a-b;`

`multiply<=a*b;` -- **ERROR**: a*b = 8 bits!

`full_multiply<=a*b;` -- full_multiply **must** have 8 bits

`divide<=a/b;`

`modulus<=a mod b;`

`add="1001" = 9`

`sub="0111" = 7`

`full_multiply="00001000" = 8`

`divide="1000" = 8`

`modulus="0000" = 0`

`end Behavioral;`

VHDL – Unsigned

Πράξεις

Έστω όλα τα σήματα unsigned(3 down to 0), 4 bits

a<="0001"; -- 1

b<="1000"; -- 8

add<=a+b;

sub<=a-b;

multiply<=a*b; -- **ERROR**: a*b = 8 bits!

full_multiply<=a*b; -- full_multiply **must** have 8 bits

divide<=a/b;

modulus<=a mod b;

add="1001" = 9

sub="1001" = **9 (Error), a-b<0**

full_multiply="00001000" --8

divide="0000" --0 (πηλίκο)

modulus="0000" --1

end Behavioral;

VHDL – Unsigned

Πρόσθεση

```
library IEEE; use IEEE.NUMERIC_STD.ALL;  
....  
signal a, b, s: unsigned (7 downto 0);  
s<=a+b;
```

+: Παράγει αποτέλεσμα ίδιου μήκους.
Δεν ανιχνεύει υπερχείλιση
Θα μπορούσαμε με προφανώς να έχουμε
και - .

Λύση
Χρειαζόμαστε ένα
επιπλέον bit
για το κρατούμενο
(Carry)

```
library IEEE; use IEEE.NUMERIC_STD.ALL;  
....  
signal a, b, s: unsigned (7 downto 0);  
signal temp_result: unsigned (7 downto 0);  
signal c: std_logic;  
  
temp_result<=("0"&a)+("0"&b);  
c<=temp_result(8)  
s<=temp_result(7 downto 0);
```

VHDL – Unsigned

Γινόμενο

- Μεγαλύτερο αποτέλεσμα για τελεστές των n bit :

$$(2^n - 1)(2^n - 1) = 2^{2n} - 2^n - 2^n + 1 = 2^{2n} - (2^{n+1} - 1)$$

- Απαιτεί 2^{2n} bit για αποφυγή υπερχείλισης
- Γινόμενο τελεστών των n bit και m bit
 - Απαιτεί $n + m$ bit

```
signal x : unsigned(7 downto 0);  
signal y : unsigned(13 downto 0);  
signal p : unsigned(21 downto 0);  
...  
p <= x * y;
```

VHDL – Unsigned

Increments

- Απλά, πρόσθεση ή αφαίρεση του 1

```
signal x, s: unsigned(15 downto 0);  
...  
  
s <= x + 1;  -- increment x  
  
s <= x - 1;  -- decrement x
```

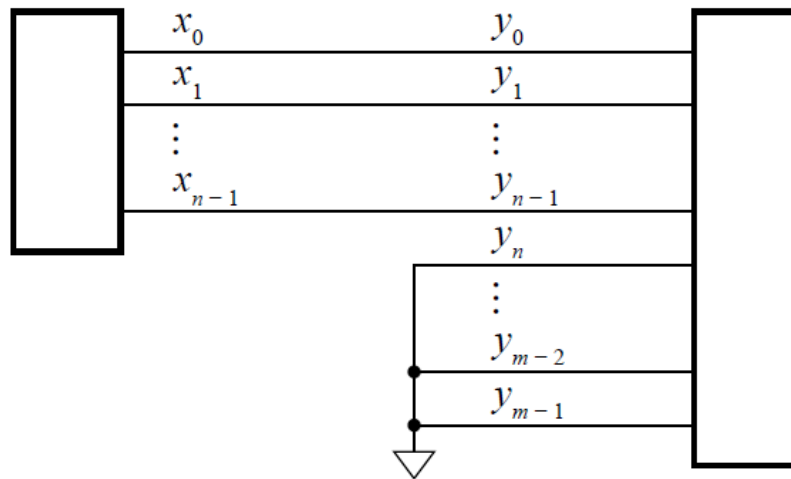
- Σημείωση: 1 (ακέραιος), όχι '1' (bit)

VHDL – Unsigned

Επέκταση μηδενός

- Για επέκταση αριθμού από n bit σε m bit
 - Πρόσθεση αρχικών bit 0
 - π.χ., $72_{10} = 1001000 = 000001001000$

Σε όλες τις εντολές όσα bit είναι στο δεξί μέλος τόσο ακριβώς πρέπει να είναι και στο αριστερό



```
signal x : unsigned(3 downto 0);  
signal y : unsigned(7 downto 0);  
y <= "0000" & x;
```

```
y <= resize(x, 8);
```

Συνένωση
-
concatenation

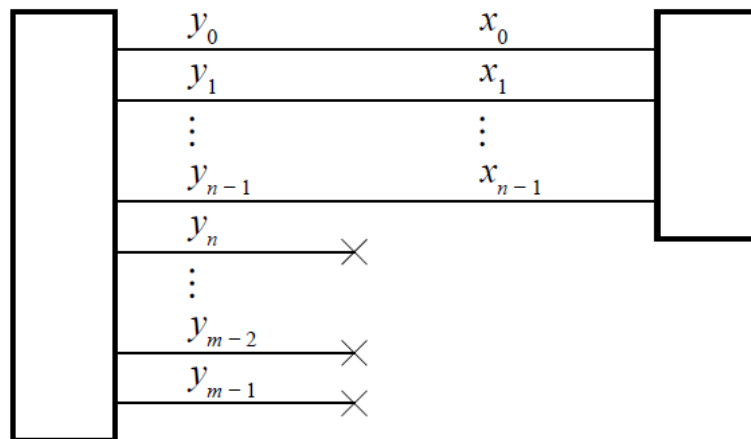
VHDL – Unsigned

Αποκοπή

Για αποκοπή από m bit σε n bit

- Απορρίπτουμε τα αριστερότερα bit
- Η τιμή διατηρείται εφόσον τα bit που απορρίπτονται είναι 0
- Το αποτέλεσμα είναι το $x \bmod 2^n$

Έστω: `signal y : unsigned(7 downto 0);`
`signal x : unsigned(3 downto 0);`



```
x <= y(3 downto 0);
```

```
x <= resize(y, 4);
```

```
x <= resize(y, x'length)
```

Ποια είναι πιο ευέλικτη;

Και για επέκταση και για αποκοπή.

VHDL – Signed

Πράξεις

Έστω όλα τα σήματα signed(3 down to 0), 4 bits

`a<="1000"`; = -8

`b<="0001"`; = 1

Οι τιμές δίνονται όπως και σε `std_logic_vector`

`add<=a+b`;

`sub<=a-b`;

`multiply<=a*b`; -- **ERROR**: $a*b = 8$ bits!

`full_multiply<=a*b`; -- full_multiply **must** have 8 bits

`divide<=a/b`;

`modulus<=a mod b`;

`absolute_value<=abs(a)`;

`add="1001"` = -7

`sub="0111"` = 7 (**Error** $a-b < -2^{(n-1)} = -8$)

`full_multiply="00001000"` = -8

`divide="1000"` = -8

`modulus="0000"` = 0

`absolute_value="1000"` = -8 (**Error** $abs(a) > 2^{(n-1)} - 1 = 7$)

VHDL – Signed

Πράξεις

Έστω όλα τα σήματα signed(3 down to 0), 4 bits

a<="0001"; = 1

b<="1000"; = -8

add<=a+b;

sub<=a-b;

multiply<=a*b; -- **ERROR**: a*b = 8 bits!

full_multiply<=a*b; -- full_multiply **must** have 8 bits

divide<=a/b;

modulus<=a mod b;

absolute_value<=abs(a);

add="1001" = -7

sub="1001" = 7 (**Error a-b>2⁽ⁿ⁻¹⁾-1= 7**)

full_multiply="00001000" = -8

divide="0000" = 0

modulus="1001" = -7

absolute_value="0001" =1

end Behavioral;

VHDL – Signed

- Τύπος `signed` από το `numeric_std`

Απαραίτητη η χρήση της βιβλιοθήκης `numeric_std`

```
library ieee; use ieee.numeric_std.all;  
...  
s : signed(15 downto 0);
```

- Οι τύποι `signed` και `unsigned` είναι ξεχωριστοί

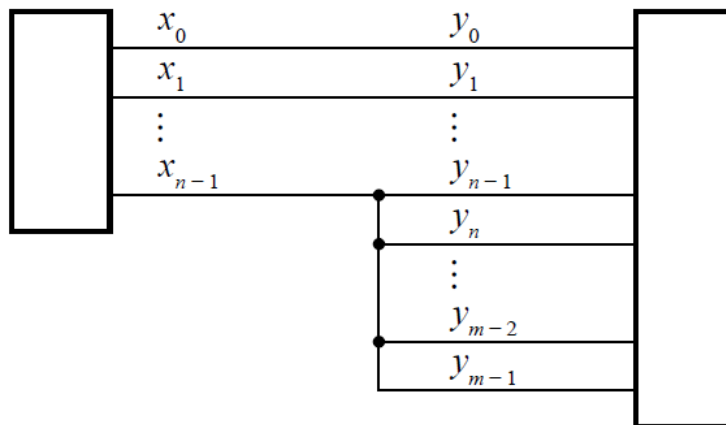
```
signal s1 : unsigned(11 downto 0);  
signal s2 : signed(11 downto 0);  
...  
s1 <= s2; -- illegal
```

```
s1 <= unsigned(s2); -- s2 is known to be non-negative  
...  
s2 <= signed(s1); -- s1 is known to be less than 2**11
```

VHDL – Signed

Αλλαγή μεγέθους

- Γενικά, για ακεραίους σε συμπλήρωμα ως προς 2
 - Επέκταση με επανάληψη του bit προσήμου
 - *Επέκταση προσήμου*
 - Αποκοπή με απόρριψη αρχικών bit
 - Τα bit που απορρίπτονται πρέπει όλα να είναι τα ίδια, και ίδια με το bit προσήμου του αποτελέσματος



```
signal x : signed (7 downto 0);  
signal y : signed (15 downto 0);  
...  
y <= resize(x, y'length);  
...  
x <= resize(y, x'length);
```

Όχι πλέον
σταθερή τιμή

VHDL – Signed

Πρόσθεση

00 0 0 0 0 0 0
72: 0 1 0 0 1 0 0 0
49: 0 0 1 1 0 0 0 1

121: 0 1 1 1 1 0 0 1

χωρίς υπερχείλιση

01 0 0 1 0 0 0
72: 0 1 0 0 1 0 0 0
105: 0 1 1 0 1 0 0 1

1 0 1 1 0 0 0 1

θετική υπερχείλιση

11 0 0 0 0 0 0
-63: 1 1 0 0 0 0 0 1
-32: 1 1 1 0 0 0 0 0

-95: 1 0 1 0 0 0 0 1

χωρίς υπερχείλιση

10 0 0 0 0 0 0
-63: 1 1 0 0 0 0 0 1
-96: 1 0 1 0 0 0 0 0

0 1 1 0 0 0 0 1

αρνητική υπερχείλιση

00 0 0 0 0 0 0
-42: 1 1 0 1 0 1 1 0
8: 0 0 0 0 1 0 0 0

-34: 1 1 0 1 1 1 1 0

χωρίς υπερχείλιση

11 1 1 1 0 0 0
42: 0 0 1 0 1 0 1 0
-8: 1 1 1 1 1 0 0 0

34: 0 0 1 0 0 0 1 0

χωρίς υπερχείλιση

VHDL – Signed

Πρόσθεση

- Το αποτέλεσμα του + έχει ίδιο μέγεθος με τους τελεστέους

Όπως ισχύει και για unsigned

```
signal v1, v2 : signed(11 downto 0);  
signal sum : signed(12 downto 0);  
...  
sum <= resize(v1, sum'length) + resize(v2, sum'length);
```

- Για έλεγχο υπερχείλισης, σύγκριση προσήμων

```
signal x, y, z: signed(7 downto 0);  
signal ovf : std_logic;  
...  
z <= x + y;  
ovf <= (not x(7) and not y(7) and z(7))  
       or (x(7) and y(7) and not z(7));
```

VHDL – Signed

Πολλαπλασιασμός

- Πολλαπλασιασμός με 2^k
 - Αριστερή λογική ολίσθηση (όπως για απρόσημους)
- Διαίρεση με 2^k
 - Δεξιά αριθμητική ολίσθηση
 - Απόρριψη των k λιγότερο σημαντικών bit, και εισαγωγή k αντιγράφων του bit προσήμου στο περισσότερο σημαντικό άκρο
 - π.χ., $s = "11110011" \text{ -- } -13$
 $\text{shift_right}(s, 2) = "11111100" \text{ -- } -13 / 2^2$

Το πρόσημο πρέπει να παραμείνει

VHDL – Τύποι σημάτων

Οι ακόλουθοι τύποι υποστηρίζονται εξ' ορισμού στην VHDL.

- bit : τιμές MONO '0' και '1' {0,1}
- Boolean: τιμές Αληθής/Ψευδής {true, false}
- Character: ASCII character (είναι 256 σε πλήθος)
- bit_vector: κατ' αναλογία με std_logic_vector
- integer: -2^{31} έως $2^{31} - 1$ (signed binary number range: $[-2^{n-1}, 2^{n-1}-1]$, 32bit word arch.)
 - Integer: range 0 to 1000
- natural: 0 έως $2^{31} - 1$
- positive: 1 έως $2^{31} - 1$

Enumerated type: Συγκεκριμένες τιμές

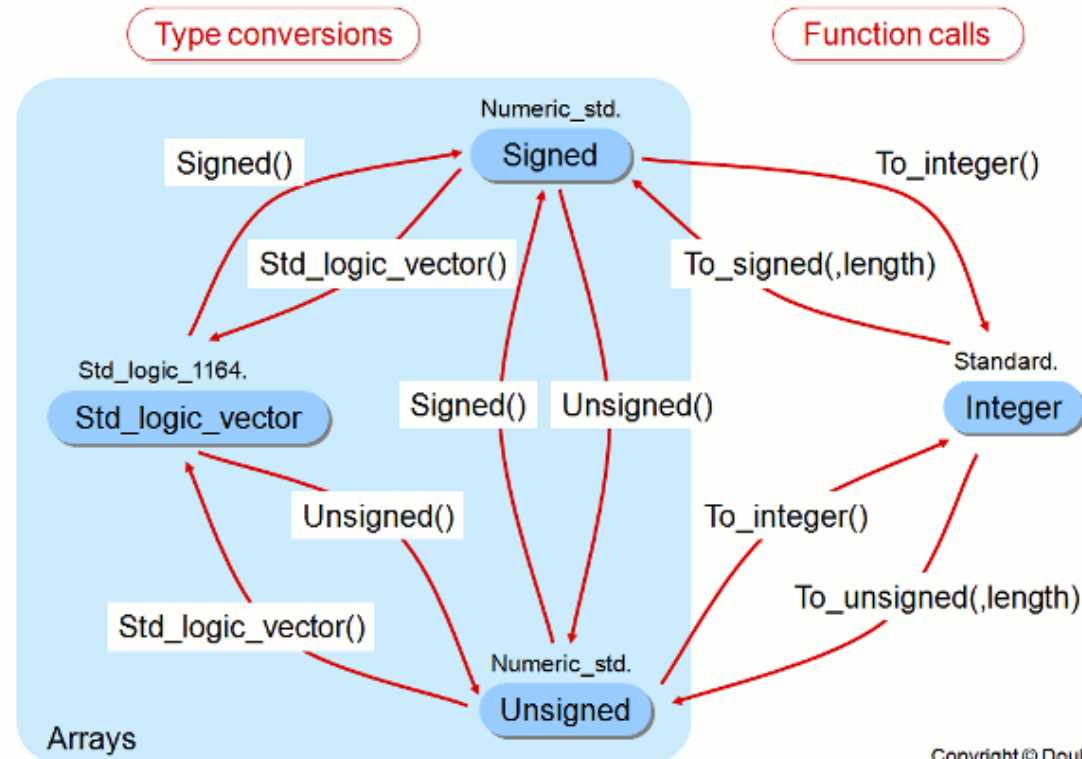
Subtypes: Υποτύποι του integer

Υπάρχουν και άλλα Type/Subtypes είτε στη standard βιβλιοθήκη είτε σε άλλες ([ieee.numeric_std.all](#): signed/unsigned, [ieee.std_logic_1164](#): std_logic, std_logic_vector)

VHDL – Τύποι σημάτων

Μετατροπές

Numeric Std Conversions



VHDL – Τύποι σημάτων

Μετατροπές

Έστω οι δηλώσεις:

```
signal x: std_logic_vector(3 downto 0);  
signal y: unsigned(3 downto 0);  
signal z: signed(3 downto 0);
```

Για να μετατρέψουμε τον ένα τύπο στον άλλο:

```
x<=std_logic_vector(y);  
ή  
y<=unsigned(x);  
ή  
z<=signed(x);  
ή  
z<=signed(y);
```

Έστω οι δηλώσεις:

```
signal x: std_logic_vector(5 downto 0);  
signal y: unsigned(3 downto 0);
```

Για να μετατρέψουμε τον ένα τύπο στον άλλο:

```
x<=std_logic_vector(resize(y,x'length));  
ή  
y<=unsigned(x(3 downto 0));
```

σωστό π.χ. και το `y<=unsigned(x(4 downto 1))`;

Όσα bit είναι αριστερά μιας εντολής, τόσα πρέπει να είναι και δεξιά

VHDL – Τύποι σημάτων

Περίπτωση τιμής 'X'

Έστω η δήλωση:

```
signal x: std_logic_vector(3 downto 0);
```

και στην αρχιτεκτονική υπάρχουν ταυτόχρονα και οι δύο εντολές:

```
x<="0101";
```

```
x<="0100";
```

Τελικά το x έχει αποτέλεσμα

```
x="010X"
```

VHDL – Τελεστές (operators)

Λογικοί Τελεστές

Operator	Operation
<code>not</code>	Logical negation
<code>and</code>	Logical AND
<code>nand</code>	Logical NAND
<code>or</code>	Logical OR
<code>nor</code>	Logical NOR
<code>xor</code>	Logical Exclusive-OR
<code>xnor</code>	Logical Exclusive-NOR

Εφαρμόζονται σε bit

VHDL – Τελεστές (operators)

Αριθμητικοί Τελεστές

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute value
**	Exponential

Εφαρμόζονται σε σήματα τύπου integer/signed/unsigned

VHDL – Τελεστές (operators)

Τελεστές σύγκρισης (Relational Operators)

Operator	Returns true if the comparison is:
=	Equal
/=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

VHDL – Τελεστές (operators)

Τελεστές Ολίσθησης (Shift/Rotate Operators)

Operator	Operation
sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left
ror	Rotate right

Εφαρμόζονται σε vector

**ΔΕΝ ΓΝΩΡΙΖΟΥΜΕ ΑΝ ΜΠΟΡΟΥΜΕ ΝΑ ΤΟΥΣ ΧΡΗΣΙΜΟΠΟΙΗΣΟΥΜΕ
(μπορούμε να τους υλοποιούμε μόνοι μας)**

VHDL – Τελεστές (operators)

Συναρτήσεις Ολίσθησης (Πακέτο numeric_std)

shift_left()
shift_right()

-

rotate_left()
rotate_right()

Αριθμητική Ολίσθηση

Εφαρμόζονται σε signed/unsigned

ΣΙΓΟΥΡΑ ΜΠΟΡΟΥΜΕ ΝΑ ΤΟΥΣ ΧΡΗΣΙΜΟΠΟΙΗΣΟΥΜΕ

VHDL – Τελεστές (operators)

Unsigned: Ολίσθηση

- Λειτουργίες `shift_left` και `shift_right`
 - Αποτέλεσμα ίδιου μεγέθους με τον τελεστέο

$s = 00010011_2 = 19_{10}$



```
y <= shift_left(s, 2);
```



$y = 01001100_2 = 76_{10}$

$s = 00010011_2 = 19_{10}$



```
y <= shift_right(s, 2);
```



$y = 00000100_2 = 4_{10}$

VHDL – Τελεστές (operators)

VHDL'93 Vivado 2019.2, **2022.2**

Για το `std_logic_vector`
ΔΕΝ υπάρχει κάποια συνάρτηση.

Εάν όμως ενεργοποιήσουμε τη
VHDL 2008 τότε μπορούμε να
χρησιμοποιήσουμε τα
`sll`, `srl`, `rol`, `ror`.

signed/unsigned

```
x<=a sll 2;  
y<=a srl 2;  
-----  
r<=shift_left(a,2);  
t<=shift_right(a,2);  
-----  
q<=a rol 2;  
u<=a ror 2;  
i<=rotate_left(a,2);  
o<=rotate_right(a,2)
```

Εάν όμως ενεργοποιήσουμε τη
VHDL 2008 τότε μπορούμε να
χρησιμοποιήσουμε και τα
`sla`, `sra`.

VHDL – Τελεστές (operators)

VHDL'93 Vivado 2022.1

std_logic_vector

```
x<=a sll 2;
```

```
y<=a srl 2;
```

```
q<=a rol 2;
```

```
u<=a ror 2;
```

signed/unsigned

```
x<=a sll 2;
```

```
y<=a srl 2;
```

```
z<=a sla 2;
```

```
w<=a sra 2;
```

```
r<=shift_left(a,2);
```

```
t<=shift_right(a,2);
```

```
q<=a rol 2;
```

```
u<=a ror 2;
```

```
i<=rotate_left(a,2);
```

```
o<=rotate_right(a,2)
```

VHDL – Τελεστές (operators)

Προτεραιότητα

	Τελεστής	Σημασία
Υψηλότερη	not	NOT
	* / mod rem	MUL, DIV, MOD, REM
	+ -	PLUS, MINUS
	rol ror srl sll	Περιστροφή, λογική ολίσθηση
	< <= > >=	Σχετική σύγκριση
	= /=	Σύγκριση ισότητας
Χαμηλότερη	and or nand nor xor xnor	Λογικές πράξεις (εκτελούνται από αριστερά προς τα δεξιά)

VHDL – Τελεστές (operators)

Μπορείτε να διαβάσετε και:

<https://www.nandland.com/vhdl/examples/example-shifts.html>

<https://redirect.cs.umbc.edu/portal/help/VHDL/operator.html>

[https://www.vhdl-online.de/vhdl reference 93/shift operators](https://www.vhdl-online.de/vhdl%20reference%2093/shift%20operators)

Μπορείτε να δείτε επίσης και τη συμπεριφορά τους σε bit_vector:

https://hdlworks.com/hdl_corner/vhdl_ref/VHDLContents/BitVector.htm

Συνοπτικός οδηγός VHDL

<https://www.ics.uci.edu/~jmoorkan/vhdlref/vhdl.html>

Για διαφορές mod/rem:

<https://stackoverflow.com/questions/25848879/difference-between-mod-and-rem-operators-in-vhdl>

VHDL – Process

- **process**: η διεργασία είναι μία ομάδα από εντολές που εκτελούνται ακολουθιακά

Sensitivity list



```
architecture arch_name of entity_name is
begin
  label: process (signal_name, ..., signal_name)
    variable variable_name: variable_type;
  begin
    sequential_statement;
    ...
    sequential_statement;
  end process;
end arch_name;
```

VHDL – Process

Variables (Μεταβλητές)

variable_name: το όνομα της μεταβλητής
(εάν είναι πολλές μεταβλητές χωρίζονται με κόμμα)

- μέσα στις διεργασίες ορίζονται **τοπικές μεταβλητές** και **OXI εσωτερικά σήματα**
- στις δηλώσεις μεταβλητών (μετά το **variable**) προσδιορίζονται μεταβλητές που μπορεί να μην έχουν τη φυσική σημασία του σήματος

variable_type: ο τύπος της μεταβλητής (STD_LOGIC)

VHDL – Process

Variables (Μεταβλητές)

```
variable my_number: natural;  
variable my_logic: std_logic:= '1';  
variable my_vector : std_logic_vector(3 downto 0):= "1110";
```

Αρχικοποίηση τιμών.
Μπορεί να γίνει
και σε ΟΛΑ τα
σήματα.

Σε process, function, procedure ...

<https://nandland.com/variables-vs-signals/>

VHDL – Process

Variables (Μεταβλητές)

```
variable_name := expression;
```

- **variable_name**: το όνομα της μεταβλητής
- **expression**: έκφραση με σήματα, μεταβλητές και τελεστές
 - στις ακολουθιακές (sequential) εντολές ανάθεσης μεταβλητής :
 - στην έκφραση προσδιορίζονται **σήματα**, που ανήκουν ή δεν ανήκουν στη λίστα ευαισθησίας, και **μεταβλητές** που δηλώνονται κατά τη δήλωση μεταβλητών
 - στο αριστερό μέρος της εντολής προσδιορίζεται **μεταβλητή** που δηλώνεται κατά τη δήλωση των μεταβλητών

ΠΡΟΣΟΧΗ
Είναι := και όχι <=

VHDL – Process

Variables (Μεταβλητές)

- Διαφορά μεταβλητής και σήματος μέσα σε μία διεργασία
 - η μεταβλητή παίρνει νέα τιμή **στιγμιαία** με τον τελεστή ανάθεσης **:=**, αμέσως μόλις εκτελεστεί η αντίστοιχη εντολή μέσα στη διεργασία
 - σε αντίθεση, το σήμα παίρνει νέα τιμή **με καθυστέρηση δέλτα δ_{delay}** , με τον τελεστή ανάθεσης **<=**, στο **τέλος** της εκτέλεσης της διεργασίας

Η χρήση των μεταβλητών μειώνει σημαντικά το χρόνο της προσομοίωσης

VHDL – Process

Διαφορά στην εφαρμογή της τιμής μίας sequential εντολής ανάθεσης μεταβλητής ή σήματος μέσα σε μία διεργασία :

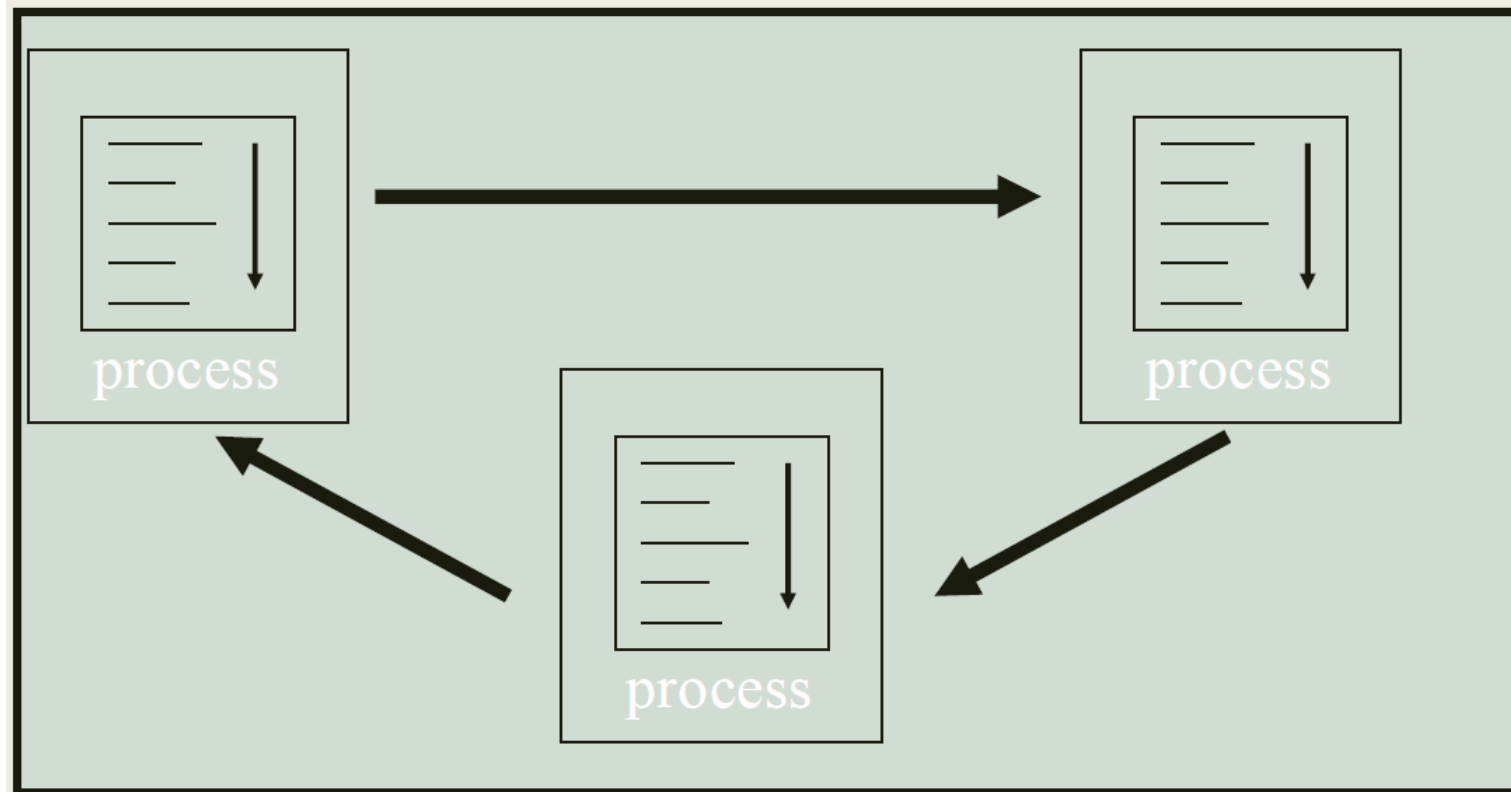
- η **μεταβλητή παίρνει νέα τιμή άμεσα** με τον τελεστή ανάθεσης :=, αμέσως μόλις εκτελεστεί η αντίστοιχη εντολή μέσα στη διεργασία
 - * Δεν χρησιμοποιείται στην υλοποίηση της λογικής
- σε αντίθεση, **το σήμα παίρνει νέα τιμή με καθυστέρηση** δέλτα δ_{delay} , με τον τελεστή ανάθεσης <=, στο τέλος της εκτέλεσης της διεργασίας
- το σήμα θυμάται την τιμή του μέχρι να φτάσει το τέλος της εκτέλεσης της διεργασίας και να λάβει μία νέα τιμή
 - * Χρησιμοποιείται στην υλοποίηση της λογικής

Signals

signal_name: το όνομα του σήματος
(εάν είναι πολλά σήματα χωρίζονται με κόμμα)

- στις δηλώσεις των σημάτων (μετά το *process*) που απαρτίζουν τη **λίστα ευαισθησίας** (*sensitivity list*) το σήμα είναι **είσοδος** της υπομονάδας που δηλώνεται κατά τη δήλωση των διαύλων της οντότητας ή **εσωτερική διασύνδεση** της υπομονάδας
- κάθε αλλαγή τιμής σήματος εισόδου που ανήκει στη **λίστα ευαισθησίας** οδηγεί στην ακολουθιακή εκτέλεση των εντολών της διεργασίας **μία φορά**
- εάν περισσότερες από μία εντολές αναθέτουν τιμή σε κάποιο σήμα λαμβάνεται υπόψη μόνο η **τελευταία ακολουθιακή εντολή**
- **Προσοχή στη δήλωση των σημάτων στη λίστα ευαισθησίας**
 - μία ελλιπής δήλωση σημάτων στη λίστα ευαισθησίας θα οδηγήσει είτε σε μη σωστή σύνθεση, είτε σε ασυμφωνία της προσομοίωσης πριν και μετά τη σύνθεση και την υλοποίηση

VHDL – Process



Κάθε διεργασία (process) εκτελεί τις εντολές της **ακολουθιακά**, ενώ πολλές διεργασίες μαζί αλληλεπιδρούν **ταυτόχρονα**. Επίσης, ταυτόχρονα αλληλεπιδρούν εντολές ταυτόχρονης ανάθεσης και διεργασίες.

VHDL – Process

- Όταν δεν υπάρχει sensitivity list οι εντολές μέσα στο process εκτελούνται συνέχεια. Θα πρέπει να υπάρχει εντολή wait ώστε να αποδοθούν τιμές στα σήματα (όπως κάνουμε στο simulation).
- Οι εντολές μέσα στο process εκτελούνται τουλάχιστον 1 φορά, ακόμα και εάν υπάρχει sensitivity list.
- Οι τιμές αποδίδονται στα σήματα είτε κάθε φορά που συναντάται η εντολή **wait** μέσα στο process είτε όταν φτάνουμε στο **end** του process.
- Εάν υπάρχει wait σε σήμα που δεν αλλάζει τιμή (εντός του process) τότε η εκτέλεση σταματάει στο συγκεκριμένο wait.
- Όλο το process θεωρείται ΜΙΑ εντολή μέσα σε μια αρχιτεκτονική και εκτελείται παράλληλα με πιθανές άλλες εντολές της.

VHDL – Process

```
test:process (a,b) is  
begin  
....  
end process test;
```



```
test:process is  
begin  
....  
wait on a,b; ← Δεν το γράφουμε αλλά ισχύει  
end process test;
```


VHDL - Παράδειγμα

Πραγματικό πρόβλημα

Σε ένα Computer Room υπάρχουν δύο (2) αισθητήρες θερμοκρασίας (**Sensor_1** και **Sensor_2**) και δύο (2) κλιματιστικά (**AirCond_1** και **AirCond_2**). Το πρώτο κλιματιστικό (AirCond_1) λειτουργεί εάν τουλάχιστον ένας από τους δύο αισθητήρες ανιχνεύσουν θερμοκρασία άνω των 35 βαθμών στο computer room. Το δεύτερο κλιματιστικό (AirCond_2) λειτουργεί εάν και οι δύο αισθητήρες ανιχνεύσουν θερμοκρασία άνω των 35 βαθμών στο computer room. Θεωρείστε ότι κάθε αισθητήρας δίνει σήμα ('1') μόνο όταν η θερμοκρασία που ανιχνεύει γίνει μεγαλύτερη των 35 βαθμών (>35). Σε άλλη περίπτωση ο αισθητήρας στέλνει την τιμή '0'. Σχεδιάστε και υλοποιήστε το λογικό κύκλωμα που περιγράφει το ανωτέρω πρόβλημα. Το όνομα του Vivado Project θα είναι Lab_1, της οντότητας θα είναι επίσης Lab_1 ενώ το όνομα της αρχιτεκτονικής Lab1 Beh.

Σχεδιάστε και υλοποιήστε το λογικό κύκλωμα που περιγράφει το ανωτέρω πρόβλημα.

VHDL - Παράδειγμα

Υλοποίηση Αρχιτεκτονικής (Dataflow)

```
architecture Dataflow of CR_AC is  
begin
```

```
    AirCond_1<=Sensor_1 or Sensor_2;  
    AirCond_2<=Sensor_2 and Sensor_3;
```

```
end architecture Dataflow;
```

VHDL - Παράδειγμα

Υλοποίηση Αρχιτεκτονικής (Dataflow)

1. Αποτελείται από απλές εντολές ανάθεσης τιμών σε σήματα
2. Κάθε εντολή εκτελείται όταν μεταβληθεί η τιμή ενός σήματος στο αριστερό μέρος.
3. Όλες οι εντολές ανάθεσης εκτελούνται ταυτόχρονα (παράλληλα)
4. Οι εντολές ανάθεσης αντιστοιχούν σε λογικές πράξεις της άλγεβρας Boole.
5. Το RTL διάγραμμα που προκύπτει είναι μία απεικόνιση της άλγεβρας Boole που εκφράζουν οι εντολές ανάθεσης σε στοιχειώδεις λογικές πύλες (AND, OR, NOT) και πολυπλέκτες.

VHDL - Παράδειγμα

Υλοποίηση Αρχιτεκτονικής (Behavioral)

```
architecture behavioral of CR_AC is  
begin
```

```
room: process (Sensor_1, Sensor_2) is  
begin
```

```
    AirCond_1<=Sensor_1 or Sensor_2;
```

```
    AirCond_2<=Sensor_1 and Sensor_2;
```

```
end process room;
```

```
end architecture Behavioral;
```

Υλοποίηση Αρχιτεκτονικής (Behavioral)

1. Όμοιο RTL Design με το Dataflow
2. ΔΕΝ χρειάζεται να αλλάξουμε τίποτα στην προσομοίωση
3. Εισαγωγή της δομής process (με λίστα ευαισθησίας)
4. Σε περίπτωση που υπάρχει μόνο η εντολή process, τότε ουσιαστικά οι εντολές μας εκτελούνται σειριακά
5. Σε περίπτωση που υπάρχουν και απλές εντολές ανάθεσης στην αρχιτεκτονική, τότε μπορείτε να θεωρήσετε όλη τη δομή process σαν μία εντολή που εκτελείται παράλληλα με τις εντολές ανάθεσης (που είναι εκτός process)

VHDL - Περίληψη

- Τύποι σημάτων Unsigned/Signed, μετατροπές τύπων
- Τελεστές
- Process
- Variable
- Παράδειγμα των αρχιτεκτονικών Dataflow και Behavioral
- Διαβάζετε τις παραγράφους 3.1, 3.2 (θεωρία και VHDL) από Ashenden και 1.4, 1.5, 2.2 - 2.5, 4.2.1, 4.2.2, 4.2.3, 4.2.6, 4.2.8, 4.2.9 (ΟΧΙ το κομμάτι της VERILOG) από το βιβλίο των Harris.
- Συνοπτικός οδηγός VHDL:
https://redirect.cs.umbc.edu/portal/help/VHDL/summary_one.html